Storage Management in a LISP-based Microprocessor

Guy Lewis Steele Jr.* and Gerald Jay Sussman**

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

* Fannie and John Hertz Fellow
** Esther and Harold E. Edgerton Associate Professor of Electrical Engineering

Abstract

We present a design for a class of computers whose "instruction sets" are based on LISP. LISP, like traditional stored-program machine languages and unlike most high-level languages, conceptually stores programs and data in the same way and explicitly allows programs to be manipulated as data. LISP is therefore a suitable language around which to design a stored-program computer architecture. LISP differs from traditional machine languages in that the program/data storage is conceptually an unordered set of linked record structures of various sizes, rather than an ordered, indexable vector of integers or bit fields of fixed size. The record structures can be organized into trees or graphs. An instruction set can be designed for programs expressed as such trees. A processor can interpret these trees in a recursive fashion, and provide automatic storage management for the record structures.

We concentrate here on the issues of memory management in such a computer, and the reasons why a layered design strategy is not only desirable and natural but even mandatory.

A prototype VLSI LISP microprocessor has been designed and fabricated for testing. It is a small-scale version of the ideas presented here, containing a sufficiently complete instruction interpreter to execute small programs, and a rudimentary storage allocator. We intend to design and fabricate a full-scale VLSI version of this architecture in 1979.

Keywords: microprocessors, large scale integration, integrated circuits, VLSI, LISP, SCHEME, list structure, garbage collection, storage management.

Introduction

An idea which has increasingly gained attention is that computer architectures should reflect specific language structures to be supported. This is an old idea; one can see features in the machines of the 1960's intended to support COBOL, FORTRAN, ALGOL, and PL/I. More recently research has been conducted into architectures to support string or array processing as in SNOBOL or APL.

An older and by now well-accepted idea is that of the stored-program computer. In such a computer the program and the data reside in the same memory; that is, the program is itself data which can be manipulated as any other data by the processor. It is this idea which allows the implementation of such powerful and incestuous software as program editors, compilers, interpreters, linking loaders, debugging systems, etc.

One of the great failings of most high-level lenguages is that they have abandoned this idea. It is extremely difficult, for example, for a PL/I (PASCAL, FORTRAN, COBOL ...) program to manipulate PL/I (PASCAL, FORTRAN, COBOL ...) programs.

On the other hand, many of these high-level languages have introduced other powerful ideas not present in standard machine languages. Among these are (1) recursively defined, nested data structures; and (2) the use of functional composition to allow programs to contain expressions as well as (or instead of) statements. The LISP language in fact has both of these features. It is unusual among high-level languages in that it also explicitly supports the stored-program idea: LISP programs are represented in a standardized way as recursively defined, nested LISP data structures. By contrast with some APL implementations, for example, which allow programs to be represented as arrays of characters, LISP also reflects the structure of program expressions in the structure of the data which represents the program. (An array of APL characters must be parsed to determine the logical structure of the APL expressions represented by the array. Similar remarks apply to SNOBOL statements represented as SNOBOL strings.)

It is for this reason that LISP is often referred to as a "high-level machine language". As with standard stored-program machine languages, programs and data are made of the same stuff. In a standard machine, however, the "stuff" is a linear (ordered) vector of fixed-size bit fields; a program is represented as an ordered sequence of bit fields (instructions) within the overall vector. In LISP, the "stuff" is a heterogenous (unordered) set of records of various sizes linked to form lists, trees, and graphs; a program is represented as a tree (a "parse tree" or "expression tree") of linked records (a subset of the overall set of records). Standard machines usually exploit the linear nature of the "stuff" through such mechanisms as indexing by additive offset and linearly advancing program counters. A computer based on LISP can similarly exploit tree structures. The counterpart of indexing is component selection; the counterpart of linear instruction execution is evaluation of expressions by recursive tree-walk.

Just as the "linear vector" stored-program-computer model leads to a variety of specific architectures, so with the "linked record" model. For concreteness we present here one specific architecture based on the linked record model which has actually been constructed.

List Structure and Programs

One of the central ideas of the LISP language is that storage management should be completely invisible to the programmer, so that he need not concern himself with the issues involved. LISP is an object-oriented language, rather than a value-oriented language. The LISP programmer does not think of variables as the objects of interest, bins in which values can be held. Instead, each data item is itself an object, which can be examined and modified, and which has an identity independent of the variable(s) used to name it.

The precise form of LISP data objects is not of concern here. The most important one, however, is the list cell (or cons cell), which is a record with two components which are pointers to other data objects; such cells are chained together by their pointers to form arbitrarily complicated graph structures. LISP provides primitive operators (CAR and CDR) following pointers, and more complex operators (ASSOC, MEMBER, PUTPROP, GET, etc.) for searching and restructuring such graphs in stylized ways.

The philosophically most important operator (CONS) effectively creates a new list cell "out of thin air". As far as the LISP programmer is concerned, new data objects are available in endless supply. They can be conveniently called forth to serve some immediate purpose and discarded when they are no longer of use. While creation is explicit, discarding is not; a data object simply disappears into limbo when the program throws away all references (direct or indirect) to that object.

The immense freedom this gives the programmer may be seen by an example taken from current experience. A sort of error message familar to most programmers is "too many nested DO loops" or "more than 200 declared arrays" or "symbol table overflow". Such messages typically arise within compilers or assemblers which were written in languages requiring data tables to be pre-allocated to some fixed length. The author of a compiler, for example, might well guess, "No one will ever use more than, say, ten nested DO loops; I'll double that for good measure, and make the nested-DO-loop-table 20 long." Inevitably, someone eventually finds some reason to write 21 nested DO loops, and finds that the compiler overflows its fixed table and issues an error message (or, worse yet, doesn't issue an error message!). On the other hand, had the compiler writer made the table 100 long or 1000 long, most of the time most of the memory space devoted to that table would be wasted.

A compiler written in LISP would be much more likely to keep a linked list of records describing each DO loop. Such a list could be grown at any time by creating a new record on demand and adding it to the list. In this way as many or as few records as needed could be accommodated.

Now one could certainly write a compiler in any language and provide such dynamic storage management with enough programming. The point is that LISP provides automatic storage management from the outset and encourages its use (in much the same way that FORTRAN provides floating-point numbers and encourages their use, even though the particular processor on which a FORTRAN program runs may or may not have floating-point hardware).

Besides providing automatic storage management, LISP provides a standardized representation for programs using standard data structures within the language. Moreover, it provides a standard operator (EVAL) which will interpret a program expressed in this standard form. This operator can itself be expressed in LISP, and the definition of such an operator constitutes a description of a processor which can execute LISP programs. This operator must be able to traverse the data structures representing the program being executed; and, because of the recursive nature of the interpretation process, needs additional temporary storage to hold intermediate results and control information. This storage can be in the form of LISP data objects.

A complete LISP system can therefore be conveniently divided into two parts: (1) a storage system, which provides an operator for the creation of new data objects and also other operators (such as pointer traversal) on those objects; and (2) a program interpreter, which executes programs expressed as data structures within the storage system. (Note that this memory/processor division characterizes the usual von Neumann architecture also. The differences occur in the nature of the processor and the memory system.)

Storage Management

Most hardware memory systems which are currently available commercially are not organized as sets of linked lists, but rather as the usual linearly-indexed vectors. (More precisely, commercially available RAMs are organized as Boolean N-cubes indexed by bit vectors. The usual practice is to impose a total ordering on the memory cells by ordering their addresses lexicographically, and then to exploit this total ordering by using indexing hardware.)

Commercially available memories are, moreover, available only in finite sizes (more's the pity). Now the free and wasteful throw-away use of data objects would cause no problem if infinite memory were available, but within a finite memory it is an ecological disaster. In order to make such memories useable to our processor we must interpose a storage manager which makes a finite vector memory appear to the evaluation mechanism to be an infinite linked-record memory. This would seem impossible, and it is; the catch is that at no time may more records be active than will fit into the finite memory actually provided. The memory is "apparently infinite" in the sense that an indefinitely large number of new records can be "created". The storage manager recycles discarded records in order to create new ones in a manner completely invisible to the LISP program interpreter.

The microprocessor we have designed therefore actually consists of two processors, side by side. One (the evaluator, EVAL) operates in terms of a LISP-style record-structure memory, and interprets LISP programs. The other (the storage manager, or garbage collector (as it is traditionally called in LISP systems), GC) serves as an intermediary between EVAL and the external memory. GC deals with the finiteness of the external memory by locating data objects which have been discarded and making them available for recycling. GC also provides EVAL with operators for dealing with the data objects.

The storage representation is a standard one using two consecutive words of memory to hold a list cell, where each word of memory can hold a pointer consisting of a type field and an address field. The address part of a pointer is in turn the address within the linear memory of the record pointed to. (This may seem obvious, but remember that until now we have been noncommittal about the precise representation of pointers, as until this point all that was necessary was that the memory system associate records with pointers by any convenient means whatsoever. The evaluator is completely unconcerned with the format or meaning of addresses; it merely accepts them from the memory system and eventually gives them back later to retrieve record components. One may think of an address as a capability for accessing a record using certain defined operations such as "fetch component number 1".)

New data objects are created in successively higher memory locations. When the finite memory is exhausted, the storage manager performs a garbage collection procedure which determines which data objects are accessible to the evaluator and which not, and compacts the former toward the low end of the memory, leaving the unused memory space in a block at the high end in which to allocate new objects.

Many techniques for garbage collection are well-documented in the literature [McCarthy 1962] [Minsky 1963] [Hart 1964] [Saunders 1964] [Schorr 1967] [Conrad 1974] [Baker 1978] [Morris 1978], and will not be discussed here. (In fact, the storage manager in the prototype we have constructed actually includes no garbage collector. The prototype was one project of a "project set" including some two dozen separate circuits, all of which had to be fit onto a single chip together. This imposed severe area limitations which restricted the address size to eight bits, and required the elimination of the microcode for the garbage collector. We anticipate no obstacles to including a garbage collector in a full-sized single-chip processor. The complexity of a simple garbage collector is comparable to that of the evaluator shown above.)

Physical Layout and Implementation

The evaluator and the storage manager are each implemented in the same way as an individual processor. Each processor has a state-machine controller and a set of registers. On each clock cycle the state-machine outputs control signals for the registers and also makes a transition to a new state.

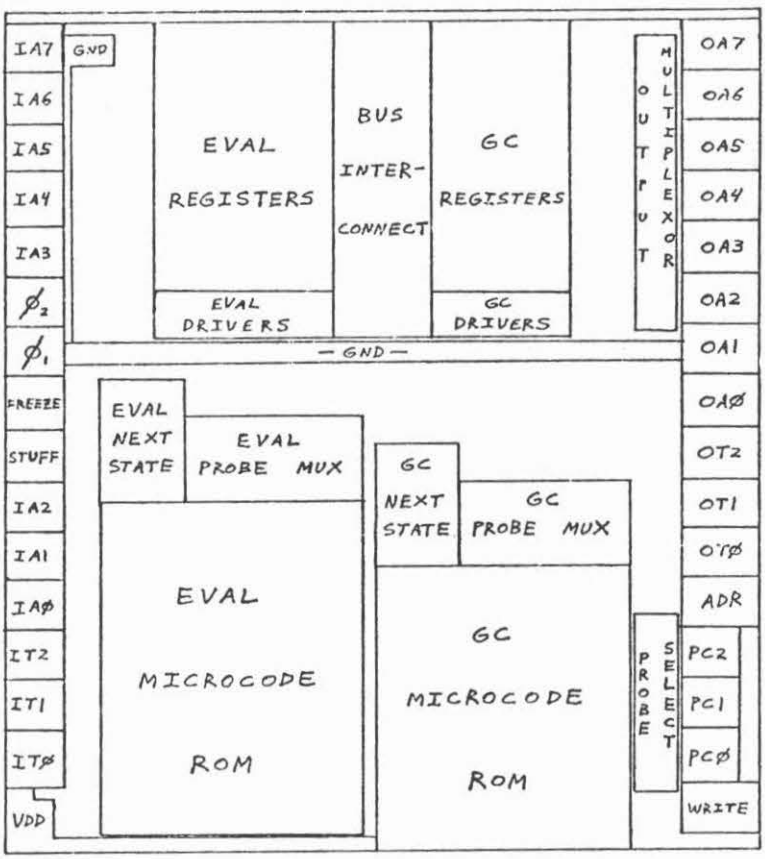
The contents of any register is a pointer, containing an address field (8 bits in the prototype) and a type field (3 bits in the prototype). The registers of a processor are connected by a common bus (E bus in the evaluator, G bus in the storage manager). Signals from the controller can read at most one register onto the bus, and load one or more other registers from the bus. One register in each controller has associated incrementation logic; the controller can cause the contents of that register, with 1 added to its address part, to be read onto the bus. The controller can also force certain constant values onto the bus rather than reading a register.

The processors can communicate with each other by causing the E and G busses to be connected. The address and type parts of the busses can be connected separately. (Typically the E bus might have its address part driven from the G bus and its type part driven by a constant supplied by the evaluator controller.) The G bus can also be connected to the address/data lines for the off-chip memory system. The storage-manager controller produces additional signals (ADR and WRITE) to control the external memory. In a similar manner, the evaluator controller produces signals which control the storage manager. (Remember that from the point of view of the evaluator, the storage manager is the memory interface!)

Each controller effectively has an extra "state register" which may be thought of as its "micro-PC". At each step the next state is computed by combining its current state with external signals in the following manner. Each "microinstruction" has a field explicitly specifying the next desired state, as well as bits specifying possible modifications of that state. If specified, external signals are logically OR'd into the desired state number. In the prototype evaluator these external signals are: (1) the type bits from the E bus; (2) a bit which is 1 iff the E bus type field is zero and a bit which is 1 iff the E bus address is zero. In the storage manager these signals are: (1) the four control bits from the evaluator controller; (2) a bit which is 1 iff the G bus address is zero. This is the way in which dispatching is achieved.

Once this new state is computed, it is passed through a three-way selector before entering the state register. The other two inputs to the selector are the current state and the data lines from the external memory system. In this way the selector control can "freeze" a controller in its current state by recirculating it, or jam an externally supplied state into the state register (both useful for debugging operations). The "freeze" mechanism is used by the storage manager to suspend the evaluator until it is ready to process the next request. In the same way, the external memory can suspend the storage manager by asserting the external FREEZE signal, thereby causing a "wait state".

(The FREEZE signal is provided as a separate control because the dynamic logic techniques usual in NMOS were used; if one stopped the processor simply by stopping the clock, the register contents would dissipate. The clocks must keep cycling in order to "refresh" the registers. The state recirculation control allows the machine to be logically stopped despite the fact that data is still circulating internally. We discovered that this technique imposed constraints on other parts of the design: the incrementation logic is the best example. It was originally intended to design an incrementing counter register, which would increment its contents in place during the recirculation of a clock cycle in which an "increment" signal was asserted. If this had been done, however, and the processor were frozen during an instruction which asserted this signal, the counter would continue to count while the processor was stopped! This could have been patched by having the FREEZE signal override the increment signal, but it was deemed simpler to adopt a design strategy in which nothing at the microcode level called for any data to be read, modified, and stored back into the same place. Thus in the actual design one must read data through modification logic and



Physical layout of prototype LISP processor

3.96 mm x 3.38 mm

156 mils x 133 mils

then onto the bus, to be stored in a different register; then if this operation is repeated many times because of the FREEZE signal it makes no difference.)

Each state-machine controller consists of a read-only memory (implemented as a programmed-logic-array), two half-registers (clocked inverters, one at each input and one at each output), and some random logic (e.g. for computing the next state). The controllers are driven by externally supplied two-phase non-overlapping clock signals; on phase 1 the registers are clocked and the next state is computed, and on phase 2 the next-state signals appear and are latched.

All of the signals from the two controllers (62 = 34+28 in the prototype) are multiplexed onto twelve probe lines by six unary probe-control signals. (These signals are derived from the three binary-encoded off-chip signals PC0-PC2.) When a probe-control signal is asserted, the memory output pads (11 data pads plus the ADR signal in the prototype) are disconnected from the G bus and connected to the twelve probe lines. In this way the chip can be frozen and then all controller outputs verified (by cycling the probe-control signals through all six states). Also recall that the controller states can be jammed into the state registers from the memory input pads. This should allow the controller microcode to be tested completely without depending on the registers and busses working.

The diagram shows the physical layout of the prototype chip. The two controllers are side by side, with the evaluator on the left and the storage manager on the right. Above each controller is the next-state logic and probe multiplexor for that controller. Above those are the register arrays, with the busses running horizontally through them. The bus connections are in the center. The input pads are on the left edge, and the output pads on the right edge. The input pads are bussed through the evaluator's register array parallel to the E bus lines, so that they can connect to the G bus. (Unfortunately, there was no time to design tri-state pads for this project.)

Discussion

A perhaps mildly astonishing feature of this computer is that it contains no arithmetic-logic unit. More precisely, it does have arithmetic and logical capabilities, but the arithmetic units can only add 1, and the logical units can only test for zero. (Logicians know that this suffices to build a "three-counter machine", which is known to be as universal (and as convenient!) as a Turing Machine. However, our LISP architecture is also universal, and considerably more convenient.)

LISP itself is so simple that the interpreter needs no arithmetic to run interesting programs (such as computing symbolic derivatives and integrals, or pattern matching). All the LISP interpreter has to do is shuffle pointers to and from memory, and occasionally dispatch on the type of a pointer. The incrementation logic is included on the chip for two reasons. In the evaluator it is used for counting down a list when looking up lexical variables in the environment; this is not really necessary, for there are alternative environment representation strategies. In the storage manager incrementation is necessary (and, in the prototype, sufficient) for imposing a

total ordering on the external memory, so as to be able to enumerate all possible addresses. The only reason for adding 1 is to get to the next memory address. (One might note that the arithmetic properties of general two-argument addition are not exploited here. Any bijective mapping from the set of external memory addresses onto itself (i.e. a permutation function) would work just fine (but the permutation should contain only one cycle if memory is not to be wasted!). For example, subtracting 1 instead of adding, or Gray-code incrementation, would do.)

This is not to say that real LISP programs do not ever use arithmetic. It is just that the LISP interpreter itself does not require binary arithmetic of the usual sort. This architecture is intended to use devices which are addressed as memory, in the same manner used by the PDP-11, for example. We envision having a set of devices on the external memory bus which do arithmetic. One would then write operands into specific "memory locations" and then read arithmetic results from others. Such devices could be very complex processors in themselves, such as specialized array or string processors. In this way the LISP computer could serve as a convenient controller for other processors, for one thing LISP does well is to provide recursive control and environment handling without much prejudice (or expertise!) as to the data being operated upon.

Expanding on this idea, one could arrange for additional signals to the external memory system from the storage manager, such as "this data item is needed (or not needed)", which would enable external processors to do their own storage management cooperatively with the LISP processor. One might imagine, for example, an APL machine which provided tremendous array processing power, controlled by a LISP interpreter specifying which operations to perform. The APL machine could manage its own array storage, using a relatively simple storage manager cued by "mark" signals from the LISP storage manager.

The possibility of additional processors aside, this architecture exhibits an interesting layered approach to machine design. One can draw boundaries at various places such that everything above the boundary is a processor which treats everything below the boundary as a memory system with certain operations. If the boundary is drawn between the evaluator and the storage manager, then everything below the boundary together constitutes a list-structure memory system. If it is drawn between the storage manager and the external memory, then everything below the boundary is the external memory. Supposing the external memory to be a cached virtual memory system, then we could draw boundaries between the cache and main memory, or between main memory and disks, and the same observation would hold. At the other end of the scale, a complex data base management system could be written in LISP, and then the entire LISP chip (plus some software, perhaps in an external ROM) would constitute a memory system for a data base query language interpreter. In this manner we have a layered series of processors, each of which provides a more sophisticated memory system to the processor above it in terms of the less sophisticated memory system below it.

Another way to say this is that we have a hierarchy of data abstractions, each implemented in terms of a more primitive one. Thus the storage manager makes a finite, linear memory look "infinite" and tree-structured. A cache system makes a large, slow memory plus a small, fast memory look like a large, fast memory.

Yet another way to view this is as a hierarchy of interpreters running in virtual machines. Each layer implements a virtual machine within which the next processor up operates.

It is important to note that we may choose any boundary and then build everything below it in hardware and everything above it in software. Our LISP system is actually quite similar to those before it, except that we have pulled the hardware boundary much higher. One can also put different layers on different chips (as with the LISP chip and its memory). We choose to put the evaluator and the storage manager on the same chip only because (a) they fit, and (b) in the planned full-scale version, the storage manager would need too many pins as a separate chip.

Each of the layers in this architecture has much the same organization: it is divided into a controller ("state machine") and a data base ("registers"). There is a reason for this. Each layer implements a memory system, and so has state; this state is contained in the data base (which may be simply a small set of references into the next memory system down). Each layer also accepts commands from the layer above it, and transforms them into commands for the layer below it; this is the task of the controller.

We have already mentioned some of the analogies between a LISP-based processor and a traditional processor. Corresponding to indexing there is component selection; corresponding to a linearly advancing program counter there is recursive tree-walk of expressions. Another analogy we might draw is to view the instruction set as consisting of variable-length instructions (whose pieces are joined by pointers rather than being arranged in sequential memory locations). Each instruction (variable reference, call to CONS, call to use function, etc.) takes a number of operands. We may loosely say that there are two addressing modes in this architecture, one being immediate data (as in a variable reference), and the other being a recursive evaluation. In the latter case, merely referring to an operand automatically calls for the execution of an entire routine to compute it!

Project History

In January 1978 one of us (Sussman) attended a course given at MIT by Charles Botchek about the problems of integrated curcuit design. There he saw pictures of processors such as 8080's which showed that half of the chip area was devoted to arithmetic and logical operations and associated data paths. On the basis of our previous work on LISP and SCHEME [Sussman 1975] [Steele 1976a] [Steele 1976b] [Steele 1977] [Steele 1978a] [Steele 1978b] it occurred to him that LISP was sufficiently simple that almost all the operations performed in a LISP interpreter are dispatches and register shuffles, and require almost no arithmetic. He concluded that if you could get rid of the ALU in a microprocessor, there would be plenty of room for a garbage collector, and one could thus get an entire LISP system onto a chip. He also realized that typed pointers could be treated as instructions, with the types treated as "opcodes" to be dispatched on by a state machine. (The idea of typed pointers came from many previous implementations of LISP-like languages, such as MUDDLE [Galley 1975], ECL [Wegbreit 1974], and the LISP

Machine [Greenblatt 1974]. However, none of these uses the types as opcodes in the evaluator. This idea stemmed from an aborted experiment in nonstandard LISP compiler design which we performed in 1976.)

In the summer of 1978 Sussman wrote a LISP interpreter based on the state machine specification. It worked.

In the fall of 1978 Lynn Conway came to MIT from Xerox PARC as a visiting professor to teach a subject (i.e. course) on VLSI design which she developed with Carver Mead of Caltech. Sussman suggested that Steele take the course "because it would be good for him" (and also because he couldn't sit in himself because of his own teaching duties). Steele decided that it might be interesting. So why not?

The course dealt with the structured design of NMOS circuits. As part of the course each student was to prepare a small project, either individually or collaboratively. (This turned out to be a great success. Some two dozen projects were submitted, and nineteen were fit together onto a single 7 mm x 10 mm project chip for fabrication by an outside semiconductor manufacturer and eventual testing by the students.)

Now Steele remembered that Sussman had claimed that a LISP processor on a chip would be simple. A scaled-down version seemed appropriate to design for a class project. Early estimates indicated that the project would occupy 2.7 mm x 3.7 mm, which would be a little large but acceptable. (The average student project was a little under 2 mm x 2 mm.) The LISP processor prototype project would have a highly regular structure, based on programmed logic array cells provided in a library as part of the course, and on a simple register cell which could be replicated. Hence the project looked feasible. Steele began the design on November 1, 1978.

The various register cells and other regular components took about a week to design. Another week was spent writing some support software in LISP, including a microassembler for the microcode PLAs; software to produce iterated structures automatically, and rotate and scale them; and an attempt to write a logic simulator (which was "completed", but never debugged, and was abandoned after three days).

The last three weeks were spent doing random interconnect of PLA's to registers and registers to pads. The main obstacle was that there was no design support software for the course other than some plotting routines. All projects had to be manually digitized and the numbers typed into computer files by keyboard (the digitization language was the Caltech Intermediate Format (CIF)). This was rather time-consuming for all the students involved.

In all the design, layout, manual digitization, and computer data entry for this project took one person (Steele) five weeks of full-time work spanning five and one-half weeks (with Thanksgiving off). This does not include the design of the precise instruction set to be used, which was done in the last week of October (and later changed!). (The typical student project also took five weeks, but presumably with somewhat less than full-time effort.)

During this time some changes to the design were made to keep the area down, for as the work progressed the parts inexorably grew by 20 microns here and 10 microns there. The number of address bits was chopped from ten to eight. A piece of logic to compare two addresses for equality (to implement the LISP EQ operation) was scrapped (this logic was to provide an additional

dispatch bit to the evaluator in the same group as the E-bus-type-zero bit and the E-bus-address-zero bit). The input pad cell provided in the library had to be redesigned to save 102 microns on width. The WRITE pad was connected to the bottom of the PLA because there was no room to route it to the top, which changed the clock phase on which the WRITE signal rose, which was compensated for by rewriting the microcode on the day the project was due (December 6, 1978). Despite these changes, the area nevertheless increased. The final design occupied 3.378 mm x 3.960 mm.

The prototype processor layout file was merged with the files for the other students' projects, and the project chip was sent out for fabrication. Samples were packaged in 40-pin DIPs and in the students' hands by mid-January 1979. As of the conference (January 22, 1979) three of the nineteen projects on the chip had been tested and found to work. The microprocessor described here will be tested in early February.

We intend to implement a full-scale version of a LISP processor in 1979, using essentially the same design strategies. The primary changes will be the introduction of a full garbage collector and an increase in the address space and number of types. We have tentatively chosen a 41-bit word, with 31 bits of address, 5 bits of type, 3 bits of "cdr code", and 2 bits for the garbage collector.

Conclusions

We have presented a general design for and a specific example of a new class of hardware processors. This model is "classical" in that it exhibits the stored-program, program-as-data idea, as well as the processor/memory dichotomy which leads to the so-called "von Neumann bottleneck" [Backus 1978]. It differs from the usual stored-program computer in organizing its memory differently, and in using an instruction set based on this memory organization. Where the usual computer treats memory as a linear vector and executes a linear instruction stream, the architecture we present treats memory as linked records, and executes a tree-shaped program by recursive expression evaluation.

The processor described here is not to be confused with the "LISP Machine" designed and built at MIT by Greenblatt and Knight [Greenblatt 1974] [Knight 1974] [LISP Machine 1977] [Weinreb 1978]. The current generation of LISP Machine is built of standard TTL logic, and its hardware is organized as a very general-purpose microprogrammed processor of the traditional kind. It has a powerful arithmetic-logic unit and a large writable control store. Almost none of the hardware is specifically designed to handle LISP code; it is the microcode which customizes it for LISP. Finally, the LISP Machine executes a compiled order code which is of the linearly-advancing-PC type; the instruction set deals with a powerful stack machine. Thus the LISP Machine may be thought of as a hybrid architecture that takes advantage of linear vector storage organization and stack organization as well as linked-list organization. In contrast, the class of processors we present here is organized purely around linked records, especially in that the instruction set is embedded in that organization. The LISP Machine is a well-engineered machine for general-purpose production use, and so uses a variety of storage-management techniques as appropriate. The processor

described here is instead intended as an illustration of the abstracted essence of a single technique, with as little additional context as possible.

We have designed and fabricated a prototype LISP-based processor. The actual hardware design and layout was done by Steele as a term project for a course on VLSI given at MIT by Lynn Conway in Fall 1978. The prototype processor has a small but complete expression evaluator, and an incomplete storage manager (everything but the garbage collector). A more complete description of the prototype processor, plus some exposition on the nature of LISP evaluators in this context, may be found in [Steele 1979]. We plan to design and fabricate by the end of 1979 a full-scale VLSI processor having a complete garbage collector, perhaps more built-in primitive operations, and a more complex storage representation (involving "CDR-coding" [Hansen 1969] [Greenblatt 1974]) for increased bit-efficiency and speed.

A final philosophical thought: it may be worth considering kinds of "stuff" other than vectors and linked records to use for representing data. For example, in LISP we generally organize the records only into trees rather than general graphs. Other storage organizations should also be explored. The crucial idea, however, is that the instruction set should then be fit into the new storage structure in some natural and interesting way, thereby representing programs in terms of the data structures. Continuing the one example, we might look for an evaluation mechanism on general graphs rather than on trees, or on whatever other storage structure we choose. Finally, the instruction set, besides being represented in terms of the data structures, must include means for manipulating those structures. Just as the usual computer has ADD and AND; just as the LISP architecture presented here must supply CAR, CDR, and CONS; so a graph architecture must provide graph manipulation primitives, etc. Following this paradigm we may discover yet other interesting architectures and interpretation mechanisms.

Acknowledgements

We are very grateful to Lynn Conway for coming to MIT, teaching the techniques for NMOS design, and providing an opportunity for us to try our ideas as part of the course project chip. The text used for the course was written by Carver Mead and Lynn Conway. Additional material was written by Bob Hon and Carlo Sequin. It should be mentioned that the course enabled a large number of students to try interesting and imaginative LSI designs as part of the project chip. This paper describes only one project of the set, but many of these student projects may have useful application in the future.

Paul Penfield and Jon Allen made all this possible by organizing the LSI design project at MIT and arranging for Charles Botchek and Lynn Conway to teach.

Charles Botchek provided our first introduction to the subject and started our wheels spinning.

The course and project chip were executed with the cooperation, generosity, and active help of the Xerox Palo Alto Research Center, Micromask Inc., and Hewlett-Packard.

Dick Lyon and Alan Bell of Xerox PARC performed plots of the projects and assembled the projects into the final mask specifications. They were of particular direct aid to Steele in debugging his project.

Glen Miranker and William Henke maintained the plotting software used at MIT to produce intermediate plots of student projects during the design cycle, and were helpful in making modifications to the software to accommodate this project.

Peter Deutsch and Fernando Corbato were kind enough to hand-carry project plots from California to Boston to help meet the project deadline.

Tom Knight and Jack Holloway provided useful suggestions and sound engineering advice, as usual. (In particular, Knight helped Steele to design a smaller pad to reduce the area of the project, and Holloway suggested the probe multiplexor technique for testing internal signals.)

This work was conducted at the MIT Artificial Intelligence Laboratory. It was supported in part by the National Science Foundation under Grant MCS77-04828, and in part by Air Force Office of Scientific Research Grant AFOSR-78-3593.

Guy Steele's graduate studies at MIT during 1978-1979 are supported by a Fannie and John Hertz Fellowship. In the spring of 1978 they were supported by a National Science Foundation Graduate Fellowship.

References

[Backus 1978] Backus, John. "Can Programming Be Liberated from the von Neumann Style? A Function Style and Its Algebra of Programs." Comm. ACM 21, 8 (August 1978),

[Baker 1978] Baker, Henry B., Jr. List Processing in Real Time on a Serial Computer. Comm. ACM 21, 4 (April 1978), 280-294.

[Berkeley 1964] Berkeley, Edmund C., and Bobrow, Daniel G. (Eds.) The Programming Language LISP: Its Operation and Applications. Information International, Inc. (Cambridge, 1964).

[Conrad 1974] Conrad, William R. A compactifying garbage collector for ECL's non-homogeneous heap. Technical Report 2-74. Center for Research in Computing Technology, Harvard U. (Cambridge, February 1974).

[Galley 1975] Galley, S.W. and Pfister, Greg. The MDL Language. Programming Technology Division Document SYS.11.01. Project MAC, MIT (Cambridge, November 1975).

[Greenblatt 1974] Greenblatt, Richard. The LISP Machine. Artificial Intelligence Working Paper 79, MIT (Cambridge, November 1974).

[Hansen 1969] Hansen, Wilfred J. "Compact List Representation: Definition, Garbage Collection, and System Implementation." Comm. ACM 12, 9 (September 1969), 499-507.

[Hart 1964] Hart, Timothy P., and Evans, Thomas G. "Notes on implementing LISP for the M-460 computer." In Berkeley and Bobrow, The Programming Language LISP, 191-203.

[Knight 1974] Knight, Tom. The CONS Microprocessor. Artificial Intelligence Working Paper 80, MIT (Cambridge, November 1974).

[LISP Machine 1977] The LISP Machine Group: Bawden, Alan; Greenblatt, Richard; Holloway, Jack; Knight, Thomas; Moon, David; and Weinreb, Daniel. LISP Machine Progress Report. AI Memo 444. MIT AI Lab (Cambridge, August 1977).

[McCarthy 1962] McCarthy, John, et al. LISP 1.5 Programmer's Manual. The MIT Press (Cambridge, 1962).

[Minsky 1963] Minsky, M. L. A LISP garbage collector using serial secondary storage. Artificial Intelligence Memo No. 58 (revised), MIT (Cambridge, December 1963).

[Morris 1978] Morris, F. Lockwood. "A Time- and Space-Efficient Garbage Compaction Algorithm." Comm. ACM 21, 8 (August 1978), 662-665.

[Saunders 1964] Saunders, Robert A. "The LISP system for the Q-32 computer." In Berkeley and Bobrow, The Programming Language LISP, 220-231.

[Schorr 1967] Schorr, H., and Waite, W. M. "An efficient machine-independent procedure for garbage collection in various list structures." Comm. ACM 10, 8 (August 1967), 501-506.

[Steele 1976a] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. LAMBDA: The Ultimate Imperative. AI Memo 353. MIT AI Lab (Cambridge, March 1976).

[Steele 1976b] Steele, Guy Lewis Jr. LAMBDA: The Ultimate Declarative. AI Memo 379. MIT AI Lab (Cambridge, November 1976).

[Steele 1977] Steele, Guy Lewis Jr. Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto. S.M. thesis. MIT (Cambridge, May 1977). operations.

[Steele 1978a] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Revised Report on SCHEME. MIT AI Memo 452 (Cambridge, January 1978).

[Steele 1978b] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two). MIT AI Memo 453 (Cambridge, January 1978).

[Steele 1979] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. Design of LISP-based Processors. MIT AI Memo, forthcoming.

[Sussman 1975] Sussman, Gerald Jay, and Steele, Guy Lewis Jr. SCHEME: An Interpreter for Extended Lambda Calculus. AI Memo 349. MIT AI Lab (Cambridge, December 1975).

[Wegbreit 1974] Wegbreit, Ben, et al. ECL Programmer's Manual. Technical Report 23-74. Center for Research in Computing Technology, Harvard U. (Cambridge, December 1974).

[Weinreb 1978] Weinreb, Daniel, and Moon, David. LISP Machine Manual (Preliminary Version). MIT AI Lab (Cambridge, November 1978).